GRADUATION PROJECT REPORT

Project:
SOLVING CARGO TRUCK LOADING PROBLEM WITH VARIABLE CHROMOSOME LENGTH GENETIC ALGORITHM

Students:
Emir Yaren Tınastepe-150315002
İbrahim Ertuğrul Seven-140315029
Mehmet Çelik- 150315047

Advisor:
Dr.Öğr.Üyesi Didem ABİDİN

To fully implement this project we started by making a small scale 2-dimensional loading problem. At this point we implemented the 3rd dimension to our problem.

2-D Cargo Loading

With 2 dimensions we will be able to see crucial points of this project before implementing to 3 dimensions. We have 20x16 cargo space to fit our boxes inside in the most efficient way.

3-D Cargo Loading

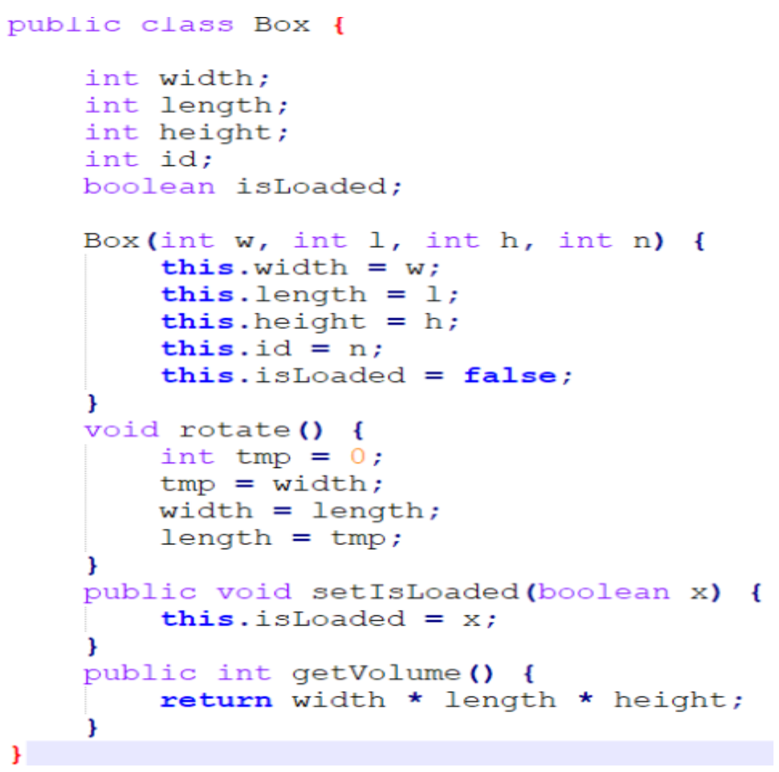
After we saw crucial point in 2-D cargo loading we implemented the 3rd dimension with the height attribute. Height is added to the container as well as the boxes. Now we have 30x10x10 cargo space to fit our 3d boxes. We can change its size as much as we want.

Boxes

The Box class was defined with 2 dimensions; width and length. After implementing the third dimension which is the height Each box has 3 dimensions. All boxes have an ID so that we don’t confuse boxes with each other while loading the cargo truck. Also each box has a boolean named ‘setIsLoaded’. This variable is false by default and it will be changed to true once it is loaded to the truck. GetVolume method is added with our new implementation. It multiplies all 3 dimensions and returns the volume of each box.

Initializing population

Population size can be set manually from its attribute. In our code we set it to 30, so we will have 30 different arraylists containing different sequences of the boxes.The boxes are generated randomly. Currently we generate 40 different boxes to be loaded. The sizes vary from 3 to 7. Each arraylist is shuffled after the boxes are set. Shuffle method is used so that we have different sequences. We use roulette wheel selection while selecting 2 genes for mutation and crossover. In a roulette wheel selection, the circular wheel is divided as described before. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.



```
public class Box {

    int width;
    int length;
    int height;
    int id;
    boolean isLoaded;

    Box(int w, int l, int h, int n) {
        this.width = w;
        this.length = l;
        this.height = h;
        this.id = n;
        this.isLoaded = false;
    }
    void rotate() {
        int tmp = 0;
        tmp = width;
        width = length;
        length = tmp;
    }
    public void setIsLoaded(boolean x) {
        this.isLoaded = x;
    }
    public int getVolume() {
        return width * length * height;
    }
}
```

Figure 1 *“Box class Code”*

Crossover operator

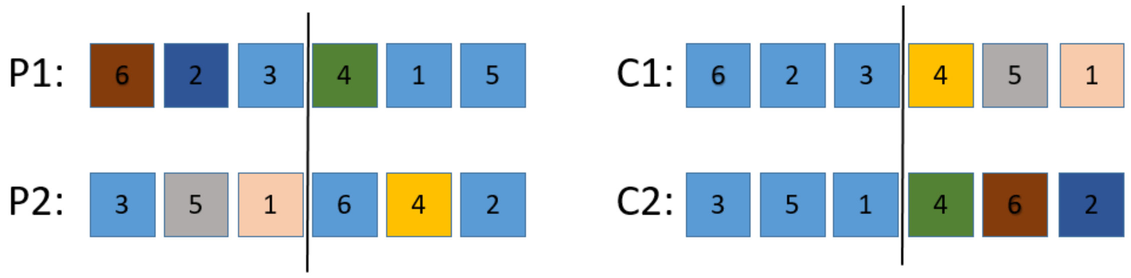


Figure 2 *“one-point crossover”*

Crossover is not changed after implementing the 3rd dimension. One-point crossover is used. Our crossover operator can be explained by this example. Let’s say that P1 and P2 are crossovered from index 2. first three indexes are passed to children without change. After these, the duplicates are ignored and the rest is passed to child in order. To fill the duplicated indexes, we loop through the parents to find the missing numbers and add them to the end of our children.

While implementing this operation we had a difficult time with the duplicated boxes that emerged after crossing. This is the best solution we could find. Our crossover method takes 2 chromosomes(arraylists) and crosses them according to the code without duplicates. To avoid duplicates we use getMissing(parent,child) method. getMissing() method finds us the missing box ID to complete our crossover operation and chromosome length.

Mutation operator

In our project we defined the mutation by rotating a random selected box in our chromosome. The boxes have a length and width if we swap these values the box will be rotated in the container. After the rotation,container is loaded again with new coordinates and fitness funcitons are calculated to determine wheter the mutation was improving or not. The rotation actually has a big impact on the fitness functions. The mutation chance can vary from %1 to %20. The next figures will show the impact of rotation by 1000th generation.With the 3rd dimension we added a swap function to our mutation which selects 2 boxes from a chromosome and swaps them.

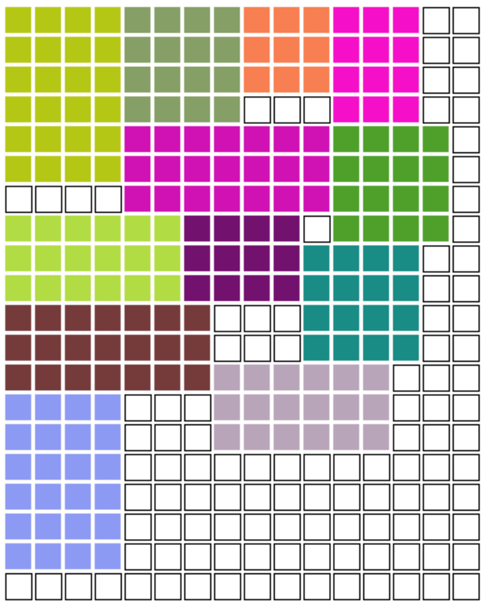


Figure 3 *" %1 chance of rotation"*

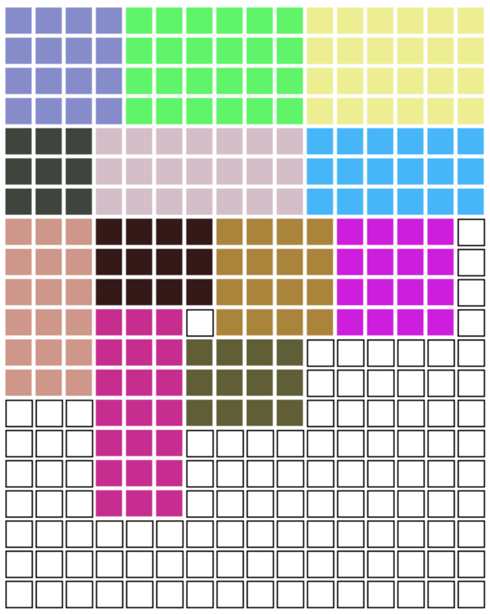
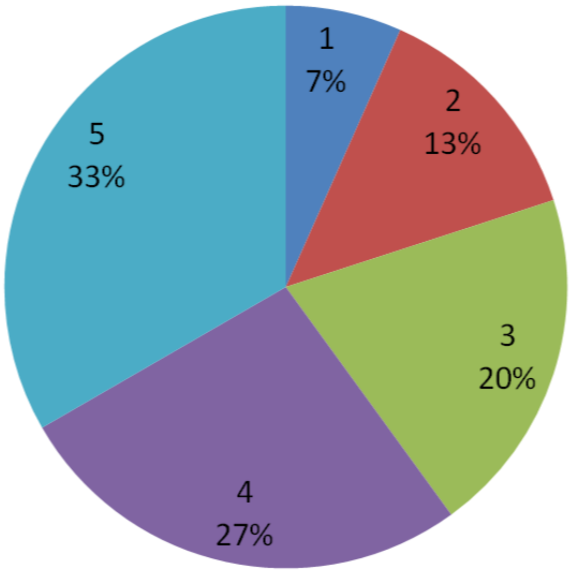


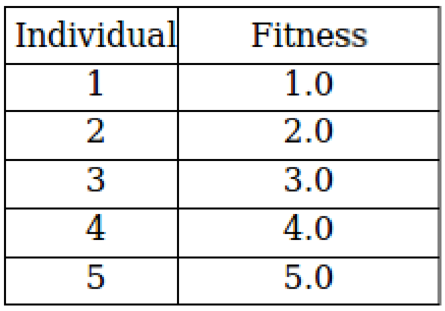
Figure 4 *" %20 chance of rotation"*

You will notice that the spaces between boxes have decreased drastically.

Roulette Wheel Selection

We sort our population according to their fitness rate after each generation. The main idea is that if a chromosome has a high fitness rate it has more probability to be selected in the next generation. And vice versa is also true. If a chromosome has low fitness rate it has a lower probability to be selected. These are determined through some complex calculations.





Individual	Fitness
1	1.0
2	2.0
3	3.0
4	4.0
5	5.0

Figure 5 *" Roulette Wheel Selection"*

Fitness Functions

We had 3 different fitness functions to determine the improvement after our mutation and crossover operations. Implementing the 3rd dimension reduced this number to only 1. We currently have one fitness function working in this code.

Total Filled Space Fitness Function

This function calculates the total remaining space after loading the container and extracts it from the total space. The less this function returns the better! We will be showing an example by what we mean about this fitness function.

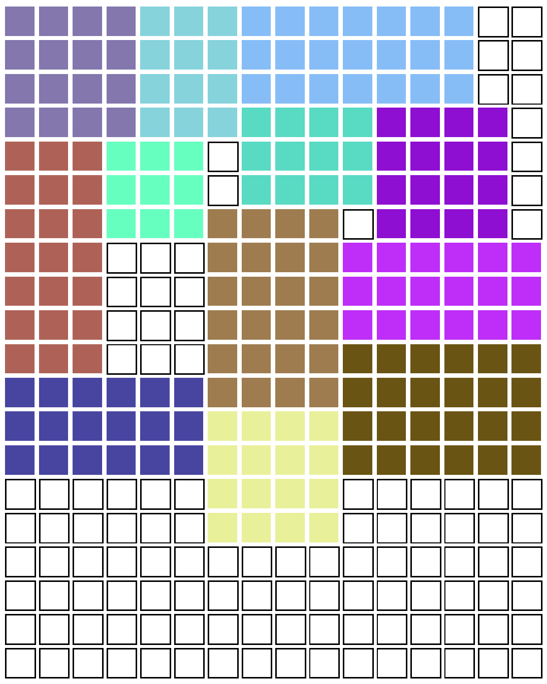
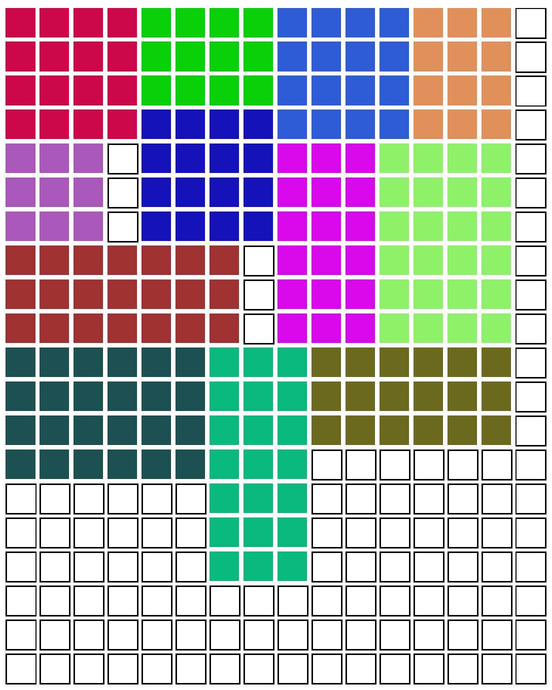



Figure 5 and 6 *"2D representation of container"*

Remaining total space: 113
Total space: 320
Fitness function=320-113=207

Remaining total space: 113
Total space: 320
Fitness function=320-113=207

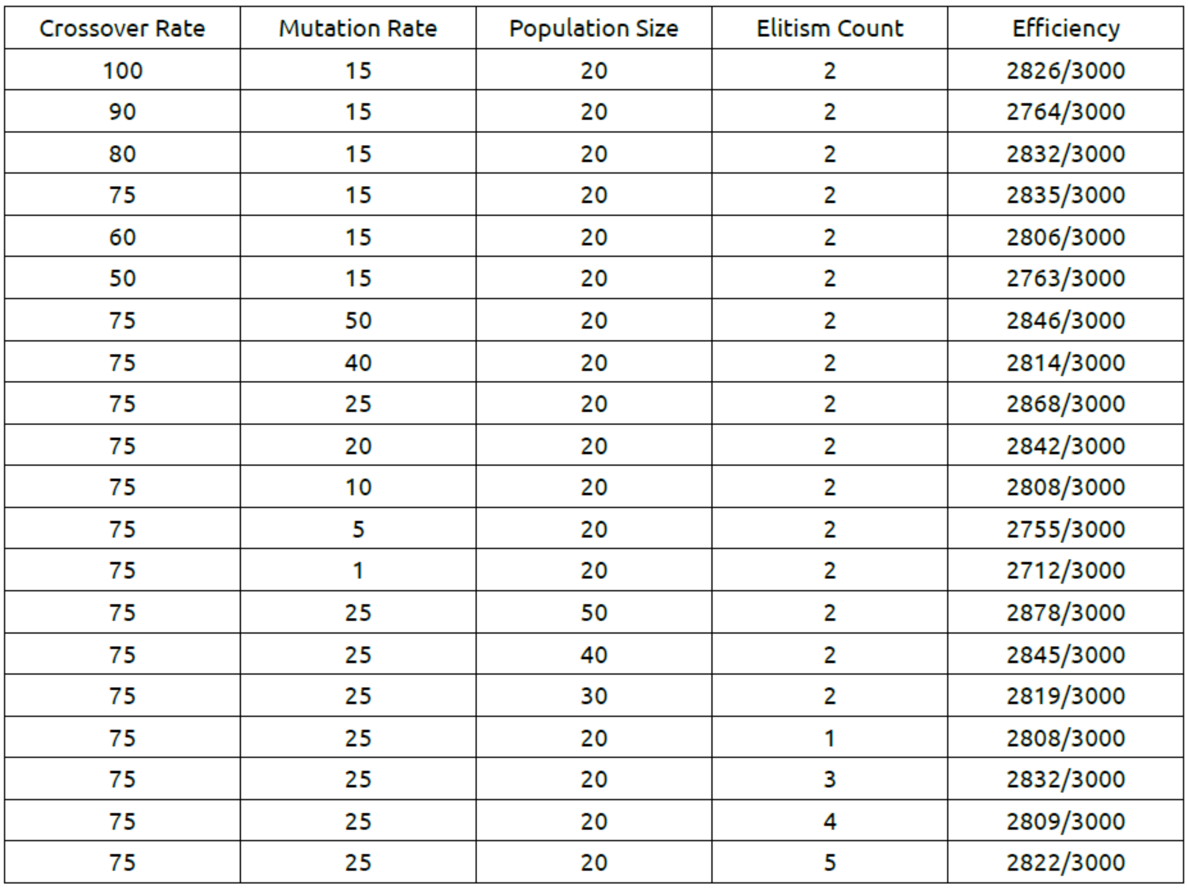
Above are the results of the first fitness function. As you can see the results are the same. We used to have 2 more fitness functions to avoid this but after implementing 3rd dimension this is no more an issue.

Elitism Count

For individuals to not lose their high fitness rates, their genes are transfered to the next generation without any change. The idea is simple; after sorting the population the highest 2 or 3 individuals are considered within this count. This can sometimes have a dramatic impact on performance by ensuring that the algorithm does not waste time re-discovering previously discarded partial solutions.

Another important topic is that we modified the algorithm so that it decides when to stop on it’s own. In our first version we limited the generation count. So it would stop at that generation. But this was not efficient and it did not generate reliable results. In our new version we again set a limit, but this limit is for the number of unimproved generations. If our limit is 100, and the fitness rate of the best solution did not improve by 100 generations, it stops. Of course setting this number too low or too high will produce either unreliable results or too much waiting time and processing costs. We think the ideal limit is around 300.

Efficiency Table



Crossover Rate	Mutation Rate	Population Size	Elitism Count	Efficiency
100	15	20	2	2826/3000
90	15	20	2	2764/3000
80	15	20	2	2832/3000
75	15	20	2	2835/3000
60	15	20	2	2806/3000
50	15	20	2	2763/3000
75	50	20	2	2846/3000
75	40	20	2	2814/3000
75	25	20	2	2868/3000
75	20	20	2	2842/3000
75	10	20	2	2808/3000
75	5	20	2	2755/3000
75	1	20	2	2712/3000
75	25	50	2	2878/3000
75	25	40	2	2845/3000
75	25	30	2	2819/3000
75	25	20	1	2808/3000
75	25	20	3	2832/3000
75	25	20	4	2809/3000
75	25	20	5	2822/3000

Figure 7 *“Efficiency Table”*

After long observations and experiments we obtained this table. Different efficiency rates are shown with different crossover and mutation rates, populations sizes and elitism counts.
With so many varying boxes and other factors it is hard to fully load the available space, but we can nearly achive 2900 with enough time and power to calculate.

3D Output

We used processing 3 for the visualization of our 3D container. Below can be seen some screenshots of the container.

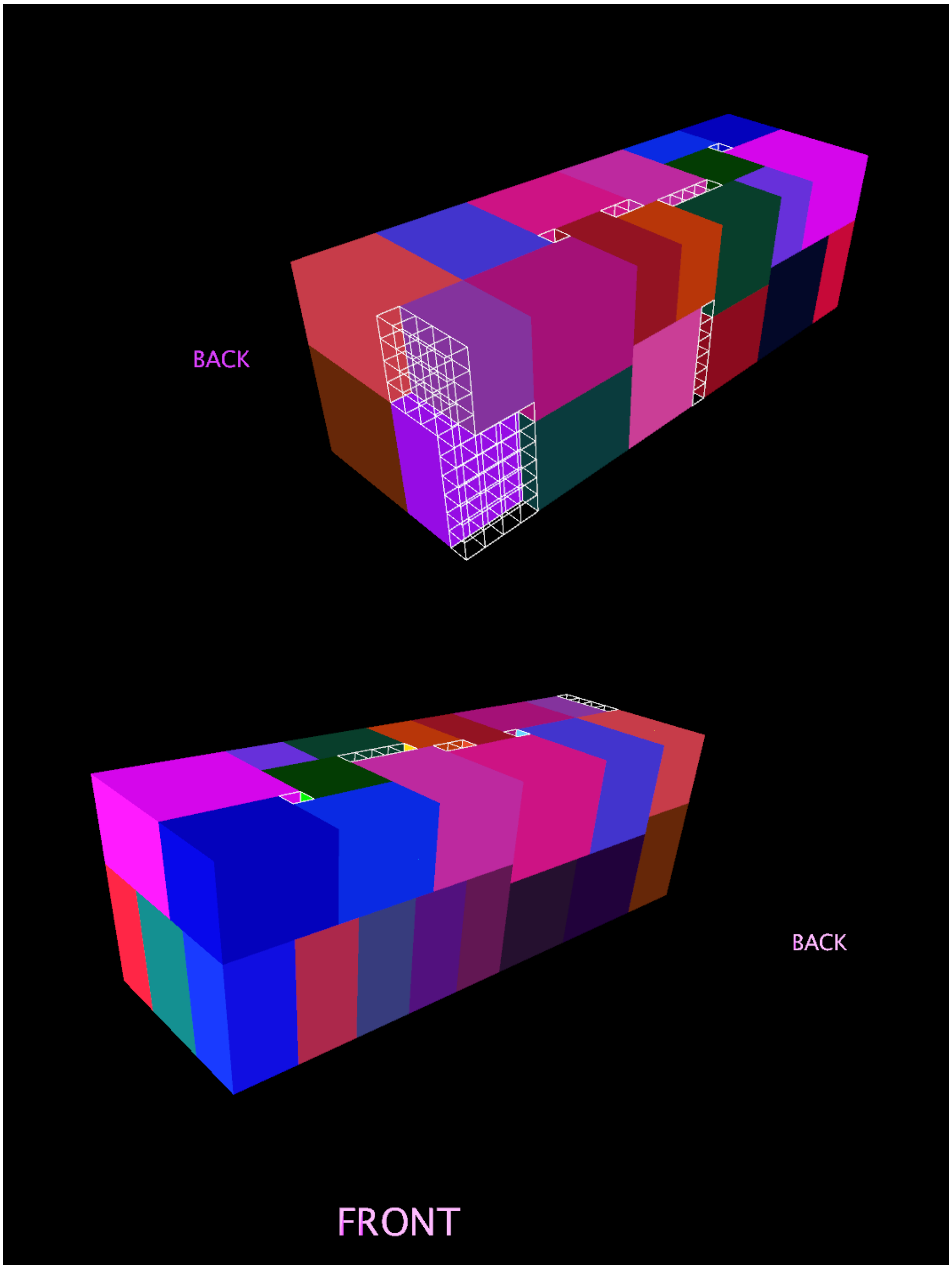
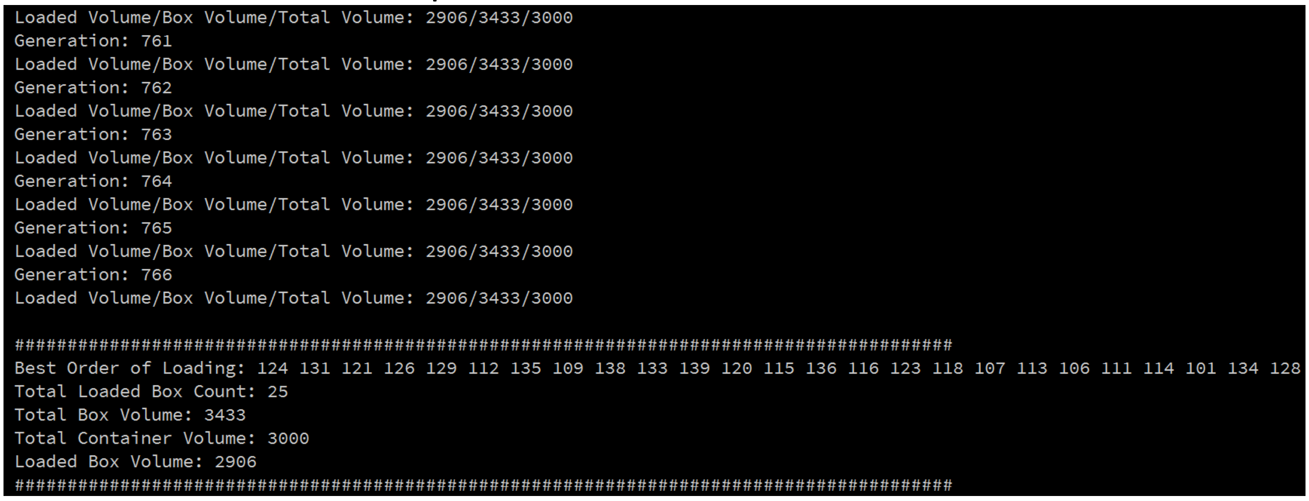


Figure 8 *"Different Angles of the loaded container"*

Terminal Output

In the terminal each generation is printed with Loaded Volume/Box Volume/Total Volume. The best solution is printed with order.Total loaded box count is also printed.



```
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000
Generation: 761
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000
Generation: 762
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000
Generation: 763
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000
Generation: 764
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000
Generation: 765
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000
Generation: 766
Loaded Volume/Box Volume/Total Volume: 2906/3433/3000

#########################################################################################
Best Order of Loading: 124 131 121 126 129 112 135 109 138 133 139 120 115 136 116 123 118 107 113 106 111 114 101 134 128
Total Loaded Box Count: 25
Total Box Volume: 3433
Total Container Volume: 3000
Loaded Box Volume: 2906
#########################################################################################
```

Figure 9 *"Terminal Output"*

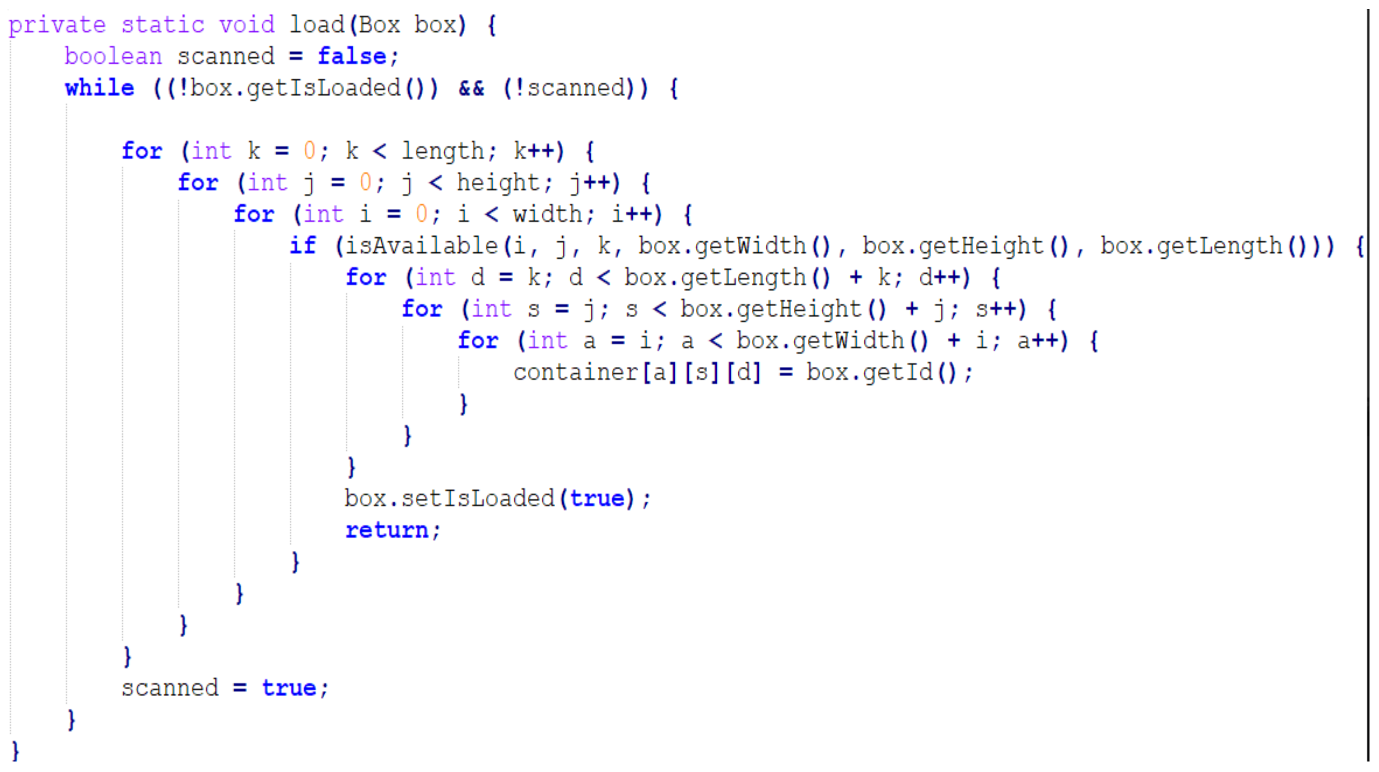
Codes

Mutation

```
private static void mutation(Individual ind) {
    Random r = new Random();
    int a1 = r.nextInt(ind.chromosome.size());
    int a2 = r.nextInt(ind.chromosome.size());
    int a3 = r.nextInt(100) + 1;

    if (a3 <= mutationRate) {
        Collections.swap(ind.chromosome, a1, a2);
        ind.chromosome.get(a1).rotate();
        ind.chromosome.get(a2).rotate();
    }
}
```

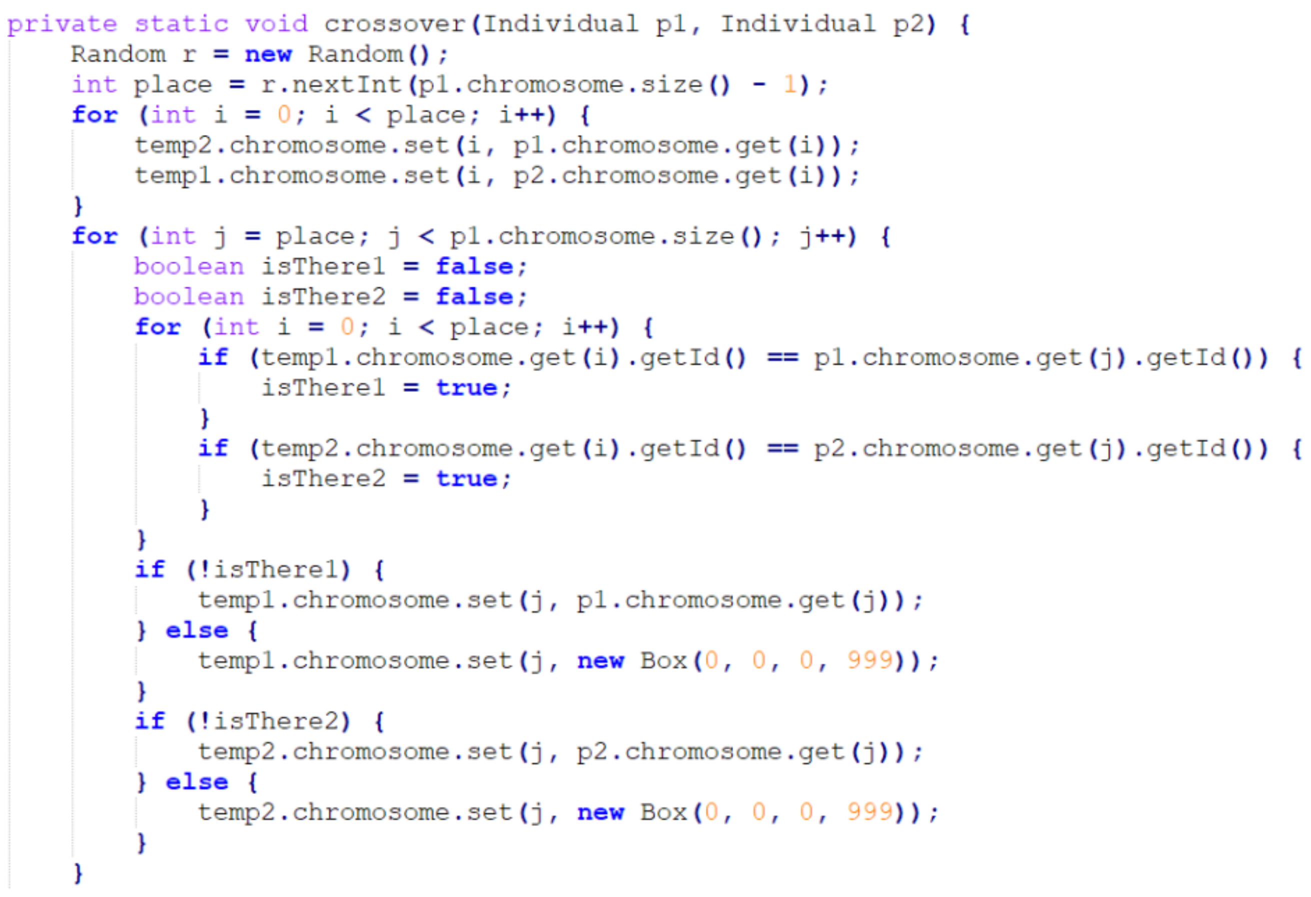
Loading to cantainer algorithm



```
private static void load(Box box) {
    boolean scanned = false;
    while ((!box.getIsLoaded()) && (!scanned)) {

        for (int k = 0; k < length; k++) {
            for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {
                    if (isAvailable(i, j, k, box.getWidth(), box.getHeight(), box.getLength())) {
                        for (int d = k; d < box.getLength() + k; d++) {
                            for (int s = j; s < box.getHeight() + j; s++) {
                                for (int a = i; a < box.getWidth() + i; a++) {
                                    container[a][s][d] = box.getId();
                                }
                            }
                        }
                        box.setIsLoaded(true);
                        return;
                    }
                }
            }
        }
        scanned = true;
    }
}
```

Crossover



```
private static void crossover(Individual p1, Individual p2) {
    Random r = new Random();
    int place = r.nextInt(p1.chromosome.size() - 1);
    for (int i = 0; i < place; i++) {
        temp2.chromosome.set(i, p1.chromosome.get(i));
        temp1.chromosome.set(i, p2.chromosome.get(i));
    }
    for (int j = place; j < p1.chromosome.size(); j++) {
        boolean isThere1 = false;
        boolean isThere2 = false;
        for (int i = 0; i < place; i++) {
            if (temp1.chromosome.get(i).getId() == p1.chromosome.get(j).getId()) {
                isThere1 = true;
            }
            if (temp2.chromosome.get(i).getId() == p2.chromosome.get(j).getId()) {
                isThere2 = true;
            }
        }
        if (!isThere1) {
            temp1.chromosome.set(j, p1.chromosome.get(j));
        } else {
            temp1.chromosome.set(j, new Box(0, 0, 0, 999));
        }
        if (!isThere2) {
            temp2.chromosome.set(j, p2.chromosome.get(j));
        } else {
            temp2.chromosome.set(j, new Box(0, 0, 0, 999));
        }
    }
    for (int i = 0; i < p1.chromosome.size(); i++) {
        if (temp1.chromosome.get(i).getId() == 999) {
            temp1.chromosome.set(i, getMissing(p1, temp1));
        }
    }
    for (int i = 0; i < p1.chromosome.size(); i++) {

        if (temp2.chromosome.get(i).getId() == 999) {
            temp2.chromosome.set(i, getMissing(p2, temp2));
        }
    }
}
```

Fitness Function

```
private static int getEfficiency(Individual individual) {
    int empty = 0;

    empty();
    for (Box box : individual.chromosome) {
        load(box);
    }
    for (int k = 0; k < length; k++) {
        for (int j = 0; j < height; j++) {
            for (int i = 0; i < width; i++) {
                if (container[i][j][k] == 100) {
                    empty++;
                }
            }
        }
    }
    empty();
    return width * height * length - empty;
}
```

getMissing

```
private static Box getMissing(Individual parent, Individual child) {
    for (Box box1 : parent.chromosome) {
        boolean isThere = false;
        for (Box box2 : child.chromosome) {
            if (box1.getId() == box2.getId()) {
                isThere = true;
            }
        }
        if (!isThere) {
            return box1;
        }
    }
    return new Box(0, 0, 0, 999);
}
```

Roulette Wheel and Sorting

```java
private static Individual roulette() {
    int a = population.size() * population.size();
    Random r = new Random();
    int b = r.nextInt(a);
    double x = Math.sqrt(b);
    int y = (int) x;
    return population.get(population.size() - y - 1);
}
```

```java
private static void sortPopulation(ArrayList<Individual> population) {
    for (int i = 0; i < population.size(); i++) {
        for (int j = 0; j < population.size(); j++) {
            if (getEfficiency(population.get(j)) < getEfficiency(population.get(i))) {
                Collections.swap(population, i, j);
            }
        }
    }
}
```

Most important codes are shown in this report, less important parts are emitted from the report.

Conclusion

In this report we tried to explain how we solved the cargo truck loading problem with variable chromosome length genetic algorithm with java. We used one-point crossover and swapping genes mutation. Elitism and roulette wheel selection is also used. Box dimensions and initial population orders are randomly decided. In our early versions we used rotating but after implementing 3rd dimension we abandoned this idea. We used processing 3 for visualizing the container. All in all we obtain very good results with our algorithm. But it can still be improved and optimized for better results and processing times.